Scheduling methods for secondary manufacturing of medicine

J.S. Zandee

May 2025

Master Thesis
Delft University of Technology

Supervisors:
A. Bishnoi, Mentor
J.T. van Essen, Graduation committee member
R.C. Kraaij, Graduation committee member

Abstract

Solving scheduling problems in practical environments, such as manufacturing facilities, can be a big challenge. Theoretical mathematical methods that aim to address these problems are oftentimes underutilized, both due to difficulties adapting them to the specific problem at hand, and limited mathematical expertise in an organization. The aim of this report is to address both of these issues by providing a step-by-step guide on the application of mathematical scheduling theory, including tools on dealing with difficult constraints. To showcase the effectiveness of this guide and its process, we provide a case study where the guide was applied to a manufacturer of generic medicine. Here we encountered many problems, such as non-standard constraints and large-sized data, but successfully addressed each of them with a mathematical model capable of generating strong, practical solutions. While further improvements are definitely possible and encouraged, this research provides a strong proof of concept on how the gap between practice and theory can be addressed.

Contents

Chapter 1

Introduction

Efficiently running a manufacturing facility is an important, but difficult task. Every day, many factors influence the production process within a facility. Some examples include workforce availability, supply chain logistics and market dynamics [1]. Even when these factors are taken into account and their effects properly predicted, unforeseen disruptions are inevitable:

- A machine may require immediate maintenance, rendering it inactive for some time.
- One of our required materials has a delay during its delivery.
- An urgent order comes through, requiring us to reallocate our resources, giving priority to this order.

Due to the dynamic nature of our production environment, making proper scheduling decisions is essential in both minimizing costs, as well as ensuring we maintain an efficient organization.

Making proper scheduling decisions is unfortunately not a trivial problem. We generally have a reasonable idea on what a good schedule should be like. We want it to make optimal use of our available resources, such as machines, workers and materials, whilst keeping costs low and meeting all our production goals. Being able to describe what makes a schedule good is an important step in our journey, but will only get us so far. Fortunately, there is a wide range of mathematical techniques that were developed precisely for tackling scheduling problems. Due to the abstract and sometimes overwhelming nature of these techniques to people without a strong basis in mathematics or engineering, many of them have only found limited use in practice [2].

This underutilization is especially a shame given the significant advantages they are able to provide to a production facility. In addition to being able to create far better schedules than one could create by hand or by using simple dispatching rules, these mathematically driven scheduling methods also provide stronger flexibility by being able to adjust schedules in response to unforeseen

circumstances. Moreover, scheduling models that make use of these techniques can assist in operational decision making by giving insight into the consequences of certain changes before they are implemented in practice.

Whilst these benefits are useful for any manufacturing facility, it is especially useful for large-scale, low-margin industries, where even small improvements in efficiency can lead to significant cost saving. One main example of this, which we often revisit during this report, is the manufacturing of generic medicine, where strict regulations, high production volumes, and intense competition influence both the importance and difficulty of efficient scheduling [3]. Other fields where scheduling methods are very important, include automotive manufacturing, food processing, and electronics production.

1.1 Objective & Scope

Having observed the current gap between theory and practice regarding mathematical scheduling theory, we would like to address this by providing a guide on how to incorporate the existing theory in practical situations. In order to achieve this goal, our aim is to develop a structured step-by-step approach to scheduling, that on top of being mathematically sound, is also practical and applicable in real-world production environments.

To demonstrate the approach we provide, the thesis includes a case study of a manufacturer of generic medicine situated in the Netherlands. More detailed information regarding this manufacturer is provided in the relevant chapters. Every step outlined in this thesis is then applied to this manufacturer, accompanied by thought processes and decision making that went into these steps. The key objectives of this research can thus be summarized as follows:

- Developing a structured step-by-step mathematical approach to real life production scheduling.
- Showcasing and reinforcing this approach by means of a case study.
- Highlighting the potential of using mathematical scheduling theory for real life applications.

1.2 Methodology & Structure

Knowing our objectives, we can identify three major questions we have to answer along the way. These questions form a nice logical path from our starting point to our objective, and are therefore an excellent basis for the structure of this thesis:

- How do we mathematically define the scheduling environment?
- How do we evaluate the quality of a schedule?
- How do we find the best schedule?

Before we can answer these questions, we need a strong background of theory that is relevant for this thesis. Therefore, we start by giving an overview of previous research related to our topic in Chapter 2. Once we have a general idea of commonly used methods and terms, we can discuss those more deeply as we answer the above questions.

With the first question we would like to explore what our scheduling environment looks like. What are the constraints and rules that a schedule has to follow? What does our input and preferred output look like? How do we translate this into mathematical terms? These questions are all addressed in Chapter 3.

In Chapter 4, we consider the second question, exploring the different options we have for scoring a schedule and how these options may affect the outcomes of our schedules.

For the fifth chapter, we take an in-depth look at the mathematical techniques out there for finding the best possible schedule in a reasonable amount of time.

For each of these three chapters, we follow a similar structure. Starting with reviewing existing theory, laying out our options, and describing the pros and cons for each of them. Next, we consider which of these options could fit well with our case study.

In Chapter 6, we perform a variety of tests in order to showcase the effectiveness of our model, as well as to finetune certain parts of it.

The last chapter consists of a conclusion.

1.3 Disclaimer

An important thought to keep in mind when reading this thesis, is that while the process is explained in a very linear way here, it is more cyclical in reality, as certain decisions we make in step 3 could affect the decisions we made in step 1 and so on.

Ultimately, this thesis aims to show that advanced mathematical scheduling methods are not just theoretical tools, but can also have a profound practical use in finding solutions to industrial problems, enhancing efficiency, flexibility, and decision-making. This work is meant as a foundation for a bridge crossing the gap between theory and practice in scheduling for production processes.

Chapter 2

Theory

Despite having found limited use in practice, there has been a lot of theoretical research on mathematical optimization for production environments. In this chapter, we discuss some of this research to give us an idea of commonly used methods and what difficulties we may encounter.

Many different modeling approaches have been applied to production scheduling and sequencing problems. Guzman et al. created an extensive review of existing research on production related optimization problems [4]. In their review, they categorized the reviewed papers based on, among other criteria, their planning horizon, model objectives and solution approach.

When considering the planning horizon, generally speaking, as we increase our horizon, the amount of uncertainty we have to deal with also increases. Dealing with this uncertainty has been done in a multitude of ways. Mula et al. classify these ways into conceptual, analytical, and simulation models [5]. With conceptual models, we do not directly model the uncertainties, but take them into account by introducing flexibility measures into our model, such as buffer times. In analytical models we directly incorporate the uncertainties in our model, for example in stochastic programming. Simulation models deal with uncertainty by simulating the uncertainty in costs, lead times, and other parameters, for example by using Monte Carlo simulation techniques.

A wide variety of solution approaches has seen use [4]. These approaches are often divided into exact methods, which guarantee an optimal solution is found (though this may take an unreasonable amount of time), and heuristic method, which do not guarantee an optimal solution, but a solution that is reasonably good, and can be obtained in a reasonable amount of time. A more detailed discussion of these methods follows in later chapters.

Besides the more general theory described above, we conclude this chapter with a brief overview of some papers dealing with a similar problem as our case study.

Ye et al. discuss a flow-shop scheduling problem with time lag consideration [6]. They minimize the makespan by first obtaining a strong solution using the permutation flow-shop problem and combine this with an iterated greedy heuris-

tic. A metaheuristic is then introduced to effectively traverse the solution space using the previously named greedy heuristic. This lead to a computationally fast solution of high quality.

Hasani et al. describe a flexible flow-shop with unrelated parallel machines and machine-dependent process stages [7]. They compare two population-based metaheuristic algorithms for optimizing both the makespan and production costs simultaneously.

Jungwattanakit et al. adapt a number of existing heuristics for the regular flow-shop to the flexible flow-shop problem and combine this with metaheuristics to optimize the positively weighted convex sum of makespan and number of tardy jobs [8].

Saidi-Mehrabad and Fattahi created a tabu search algorithm for the flexible job shop problem, showcasing that this algorithm can produce optimal solutions for small and medium sized problems in a short amount of time, and produce useful solutions for large size problems [9].

Chapter 3

Problem Definition

Before we can answer the question of what our specific scheduling environment looks like, we should get an idea of what scheduling environments look like in general. We use a number of slightly abstract definitions and mathematical notations to characterize different scheduling environments, but before introducing any of these notations, we start with an example (purposefully oversimplified) scheduling problem written in plain english. Afterwards we gradually introduce mathematical definitions and notations before ending this section with reintroducing the same problem in mathematical terms.

3.1 Example: Running a Restaurant

Suppose we are running a small local restaurant where we are serving a single group of five customers tonight. After taking their orders, we now know that one of them ordered a pizza, two of them ordered our signature pasta dish, one ordered risotto and the last customer ordered a simple salad.

In our kitchen we have two skilled chefs available, both of them capable of taking on any of the dishes at the same speed as the other. The time it takes for one of the chefs to prepare a dish differs for each dish and can be summarized as follows:

- Salad: 15 minutes.
- Pizza: 20 minutes.
- Pasta: 25 minutes.
- Risotto: 35 minutes.

We obviously wish to serve our customers as quickly as possible, since they are hungry and would prefer their food faster rather than later. Ideally, we also want to serve all 5 dishes at the same time, making sure there are no group

members still waiting for their food while the others are already enjoying their meals. Our goal is then to minimize the time that it takes for all dishes to be prepared. In this case, we do not particularly care when individual dishes are done, but instead when the collective is finished.

We can try to solve this particular scheduling problem by hand. A great starting point would be looking at the total time it would take for one chef to make all the dishes, which brings us to a total of:

$$15 + 20 + 2 \cdot 25 + 35 = 120$$

Divide that by the number of chefs we have and we find that the average workload per chef is $\frac{120}{2} = 60$ minutes. The best possible solution would then be one where each of the chefs is busy for exactly 60 minutes, and we can serve all our customers after 1 hour. If one of the chefs would be busy for less than 60 minutes, the other would have to be busy for more, meaning our collective completion time would be larger than 60 minutes, resulting in us being able to serve our customers later. In our example, we can accomplish this by having chef 1 preparing the risotto and one of the pastas, and chef 2 preparing the other dishes. Both chefs would then be busy for 60 minutes, thus giving us an optimal schedule.

Note that it is not always possible to attain this theoretical optimum where each chef is busy for exactly the average workload. For example, consider all our dishes taking 20 minutes to prepare. The average workload would be

$$\frac{5 \cdot 20}{2} = 50$$

minutes, but a single chef could never choose a number of dishes whose preparation times add up to 50. We dive more deeply in to the difficulties with finding good solutions in Chapter 5.

A written description of a solution, such as the one above, may be sufficient for relatively small cases, but as the size of a problem increases, so does the need for a more visual descriptive method that allows for quick assessment and comparison of potential solutions. A commonly used method for this, is the use of Gantt-charts. An example corresponding to our restaurant example we discussed, can be found in Figure 2.1.

3.2 Basic Definitions

Scheduling problems may arise in many broad and varied fields. We have seen it come up in a restaurant in our example above, where we were talking about chefs and dishes. It may also come up in hospitals, where we are scheduling patients in operating rooms. Many more examples exist and most of them have entirely different things we are scheduling (dishes, patients, etc.) and places where they are scheduled (chefs, operating rooms, etc.). In order to have some uniformity between similar problems in varying fields, we adopt some of the common naming conventions.

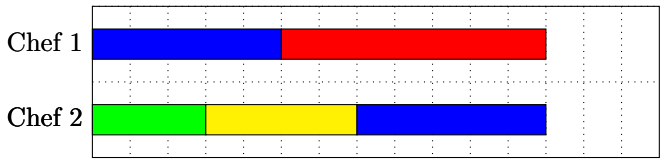


Figure 3.1: Here you can see a Gantt-chart corresponding to our restaurant. Each horizontal bar represents one dish a chef is cooking, where the length of the bar is determined by the preparation time of the dish. In this example, chef 1 prepares one pasta and one risotto, and chef 2 prepares a salad, pizza, and pasta.

In scheduling theory we usually refer to *jobs* and *machines*. Jobs are the things or tasks we want to schedule. They have a certain *processing time*, which is the time that it takes a machine to process the particular job. Machines are then the things that process the tasks. So in our restaurant example, the dishes we have to make are the jobs and the chefs that make them are the machines.

Now that we have our basic definitions, we can look a bit more into how scheduling problems are often classified, based on certain rules and characteristics of the specific problem.

3.3 Scheduling Classification

While applications of scheduling theory to different fields share some broad common characteristics, they also differ in various more specific aspects. In order to efficiently capture these intricacies, mathematicians devised a classification scheme with which one can quickly describe a scheduling environment without having to explain the whole problem. To give an idea of what kind of intricacies and characteristics we are dealing with, we take our restaurant example and address some potential problems with the current model, before we actually give the classification scheme.

For our restaurant we made a number of explicit and implicit assumptions that may or may not reflect reality. Some of those may have already crossed your mind whilst reading the example. But in order to get the point across, here follows a non-exhaustive list of reasons why the given model would most likely fail miserably in practice:

- Surely making two servings of pasta would not take twice as long as a single serving.
- A single dish can be partially prepared by both chefs.
- One chef can work on multiple dishes simultaneously; while the pizza is in the oven, the chef can work on another dish.
- The two chefs are most likely not exactly as fast as one another.

Of course we are never able to perfectly model a scheduling problem, taking all real-life factors into account, as there are simply way too many. But it is important that the ones that we do choose to ignore, are at least minor enough that they do not have a profound impact on the final results. If the chefs are of roughly equal skill, it would not be disastrous to assume they are exactly as fast as one another. Not taking multi-tasking into account has a greater impact however.

3.3.1 The Scheme

The characteristics that we are taking into account can generally be divided into three categories. They either describe the *machine environment*, for example when talking about whether our chefs prepare dishes at the same speed. They may also describe the *job characteristics and constraints*, for example whether a single dish can be divided between multiple chefs. Lastly, there may be characteristics that describe the *objective function* which is the thing we are trying to optimize. In our example the objective function was the minimization of the collective completion time of all dishes. These categories are then combined and expressed as $\alpha \mid \beta \mid \gamma$ [10], where:

- α describes the machine environment.
- β describes the job characteristics and constraints.
- γ describes the objective function.

In this chapter we limit ourselves to α and β, focusing on the rules and constraints that define a schedule. In Chapter 4, we look more closely at γ.

3.3.2 Machine Environment

The machine environment describes how jobs interact with available machines. It can be broadly divided into three categories: single-machine systems, parallel-machine systems, and multi-stage scheduling, the last of which is also known as shop scheduling.

In single-machine systems, denoted by $\alpha = 1$ in the three-field notation we adopted, all our jobs are sequentially processed by a single machine. Many of the problems in this category are relatively easy and for example solvable by following one or two simple rules when choosing the order in which we want our jobs to be processed. Even in this simple environment however, there exist very difficult problems to solve, even with the help of computers [11].

In parallel-machine systems, we have multiple machines that can work in parallel, meaning each job needs to be processed by only one of the machines. We can further divide this category into three cases, given with their relevant notation:

- $\alpha = Pm$ when dealing with identical machines, meaning that all machines operate at the same speed.

- $\alpha = Qm$ when dealing with uniform machines, where machines may have different speeds, but the overall processing works in the same manner. So one machine may be 10% faster overall than another.
- $\alpha = Rm$ denotes unrelated machines. Here every machine may have its own unique efficiencies. Returning to our restaurant example: Chef 1 may be faster at preparing pastas than chef 2, but chef 2 is faster at preparing risotto. There is no constant factor between their speeds.

For each of the above notations, the m is used to describe the number of parallel machines that are used. Our restaurant with 2 chefs of the same speed would thus be $\alpha = P2$.

Lastly, in multi-stage scheduling jobs may undergo multiple processing stages, where each stage can have its own rules and system. This category can also be further divided in three cases:

- $\alpha = O$ in an open-shop problem, where every job needs to visit a number of stages, but the order in which these stages are visited is flexible.
- $\alpha = F$ in flow-shop scheduling, where the order in which the stages are visited is fixed and the same for every job.
- $\alpha = J$ in job-shop scheduling, where the order in which the stages are visited is also fixed, but may differ between jobs. So for each job, there is only one correct order, but that order is not necessarily the same for each job.

3.3.3 Job Characteristics and Constraints

The job characteristics and constraints are denoted by β. Any rules or constraints pertaining to how jobs can be timed and ordered are found here. Unlike with the machine environment, more than one option can be picked and combined for β. Common job-related factors include:

- $\beta = r_j$: Each job has a corresponding *release date*, which specifies the time at which the job becomes available for processing.
- $\beta = d_j$: Similar to release dates, we also have due dates, which specify the time at which a job must have been completed. This constraint is often used in conjunction with certain objectives, where we would aim to schedule in such a way that few jobs are completed after their due date.
- $\beta = pmtn$: *Preemption* determines whether a job can be interrupted and resumed later.
- $\beta = prec$: Some jobs have *precedence* relations, meaning that some jobs can only be started after others have been completed.

- $\beta = s_{ij}$: *Sequence-dependent setup times* refer to the time that is required to reconfigure machines and is dependent on the sequence of the jobs.
- $\beta = no - idle$: When the no-idle rule is active, machines must operate continuously without any idle periods between tasks.

Many more of these constraints exist, reflecting all kinds of rules that may arise in practical settings.

With this basis in mind, we can put it to the test with our case study.

3.4 Case Study

Before we apply the above mentioned theory to our case study in order to classify it as a scheduling problem, we first expand a bit more upon our manufacturer. As mentioned before, our manufacturer concerns itself with the secondary manufacturing of generic medicine. What exactly does this entail?

The manufacturing process of medicine can be divided into two main parts. The first part, named primary manufacturing, entails the creation of the active pharmaceutical ingredient (API), which is the part of the medicine that has the desired chemical properties in order to fulfill its function as medicine. The second part, secondary manufacturing, consists of combining the (API) with other ingredients, called excipients, that are less about the chemical function of the medicine and more about secondary properties, such as the physical qualities of the tablets. After combining these ingredients, they are taken all the way into a packaged product, ready for consumption.

When referring to generic medicine, we mean pharmaceutical drugs that contain the same API as a brand-name equivalent. This means they essentially have the same quality, safety and performance as the 'original' drug. These generics are generally sold at a substantially lower price as the branded alternative, as they did not require the extensive research, development, and marketing costs associated with branded drugs. Once the patent of the original expires, other manufacturers can produce generics, increasing accessibility and lowering prices for patients. Due to lower prices and increased competition, production of generics is a low-margin business [3].

Our specific manufacturer has limited its product to drugs in tablet form, meaning most products share a similar production process. We can summarize the production process of our manufacturer, starting when an order is placed by a client, by the following steps:

1. Purchase & acquisition of API and excipients.
2. Substances are tested by the quality department (QD).
3. Weighing of substances.
4. Mixing & granulation of substances.
5. Granulate gets pressed into tablets.

6. Tablets receive a coating (not applicable to all products).
7. Tablets are tested by QD.
8. Tablets are packaged.
9. Final products are tested by QD.

After the last step, the products are ready for collection by the client. In order to increase the efficiency of the overall process, all of these steps need to be streamlined. We go over each of the steps in a bit more detail now.

The first step, acquisition of API and excipients, is subject to considerable uncertainty, due to long delivery times and varying supply. One solution to this problem would be for a manufacturer to stock up on API and excipients, reducing the effect of a missed delivery. Unfortunately, this is not feasible in practice for many products, due to the varying demand for medicine. Stocking up on a certain product, only for it to expire as a result of reduced demand, would lead to extreme costs that the manufacturer cannot deal with. Practically speaking, this results in the manufacturer purchasing the substances as soon as a client confirms an order. Any disturbances in the acquisition process are then taken into account by updating the expected starting time of the corresponding product.

Steps 2, 7 and 9 consist of the quality department testing the product, for example by taking samples and putting these through a variety of tests in the laboratory. This process is to ensure that nothing went wrong in the production process and the medicine has its desired effect. The turnaround time for these processes are generally speaking very reliable. This is true both when the product passes all tests, as well as when there are some deviations, as the follow-up procedures are very well-documented.

Lastly, steps 3-6 and 8 make up the actual manufacturing of the products. These production stages consist of a number of (possible different) machines per stage, where the products can be processed. Each product needs to be processed by a single machine at every stage (with some exceptions e.g. not all tablets need a coating) and every machine can only process a single product at any given time. Not all products can be processed by every machine and when multiple machine alternatives are allowed, the processing time may differ for that product.

Aside from the time spent processing products, our machines also require some setup time, in which certain settings may be adjusted and the machine is cleaned. The time spent cleaning is definitely significant, as this is a thorough process due to the high health and quality standards for manufacturing of medicine. Depending on the product that was processed before, the cleaning process can be slightly relaxed, resulting is sequence-dependent setup times (e.g. when the current product is similar to the previous product, cross-contamination is less of a concern). For now, we differentiate between two types of cleaning. A wet cleaning, which is very thorough and takes a longer time, and a dry cleaning, which is slightly relaxed and takes less time.

3.5 The First Model

In an ideal world, we have now identified all important aspects of our scheduling problem and included all important rules that our schedule needs to follow. Realistically we are bound to forget something and also want to prevent ourselves overcomplicating the first model. An all-encompassing model may be better in theory, but as we might run into computational problem later on, it is not a bad idea to skip some more niche constraints for now. Either way, we are now ready to properly define our problem. Using the three-field notation we introduced before as our guideline, we continue by considering what our machine environment and job characteristics and constraints look like.

For the machine environment, we clearly are dealing with a multi-stage system, where most products follow a similar path. Even with the exceptions that exist, such as the coating stage not being necessary for all products, we can consider our environment to be a flow shop. Each product follows the same path after all, except some of them being able to skip a certain stage. We can accomplish this mathematically by specifying that product having a processing time of 0 at the skipped stage.

Within each stage, we are also dealing with multiple machines. These machines do have some relations with each other, but they are not close to identical or uniform. We can thus describe each individual stage as an unrelated parallel-machine system. This brings us to $\alpha = F$ in our three-field notation, with the additional knowledge that the individual stages have $\alpha = Rm$, with the number of machines varying at each stage.

For the job characteristics and constraints, we currently have two rules that have a great impact on our schedules. We are dealing with sequence-dependent setup times and our jobs have release dates. This brings us to $\beta = r_j, S_{ij}$.

For now, our scheduling problem thus looks like $F \mid r_j, S_{ij} \mid \gamma$. In the next chapter, we continue by also filling in for γ.

Chapter 4

Objective Function

Now that we have a general idea of what our schedule looks like in terms of jobs and machines, and the rules that are affecting our schedule, it is time to start looking in to what we are trying to optimize.

Suppose we have two different schedules in front of us. How do we decide which of these is better? Sometimes this is very clear, in our restaurant example we simply looked at what time we could serve the entire table of customers, and would pick the schedule that minimizes this. What if two different schedules both would be able to serve the customers after 60 minutes? Do we just pick either, or do we have other criteria to prefer one over the other? What if instead of our singular table with 5 customers, we are running a large restaurant with 30 tables? Deciding between 2 different schedules will be a lot more difficult in that case.

It is clear that it is not always immediately obvious what makes one schedule better than another. One solution for this problem is to give each schedule a score, quantifying how good it is. This score is calculated by one or more mathematical functions that reflect qualities we like or dislike about solutions. When running our 30-table restaurant, we may choose to score a schedule based on the number of tables we serve within 45 minutes of them taking an order. This can easily be computed for a given schedule and gives us an easy way to compare them. Whether the schedule that performs the best for this scoring function is also the best schedule remains to be seen however. Perhaps 25 tables can be served within 45 minutes, but the other 5 take over 3 hours, while another optional schedule would have had all 30 tables served within 50 minutes of them taking their orders.

We can see that careful choices regarding how we score schedules are very important, as they directly influence what schedules will and will not be considered as top candidates. We continue by introducing some of the commonly used definitions regarding these scoring functions as well as some common examples of them.

4.1 Commonly Encountered Object

The score functions we are talking about are commonly referred to as the *objective function* of a scheduling problem, which we shortly touched upon when introducing the three-field notation in last chapter. The score that a schedule gets from the objective function is referred to as the *objective value*. Depending on the context of our objective function, we may be looking for the schedule with the highest objective value, or the lowest. Either way, we refer to the 'best' objective value as the *optimal value* and the corresponding schedule as an *optimal solution*. Note that it is *an* optimal solution, since it may not necessarily be unique.

Common examples of objective functions include:

- $\gamma = C_{max}$: C_j refers to the completion time of a job. $C_{\max}$ is then the maximum completion time, or the time at which the last job is finished, like in our singular table restaurant example. This objective function is used when all we care about is when our entire process is finished. We do not care about when certain parts are finished, just when the collective is done. This objective function is known as the makespan, or makespan minimization.

- $\gamma = \sum C_j$: With this objective function, known as total completion time, we wish to minimize the sum of all completion times. It makes sure that individual results are considered, instead of only the collective.

- $\gamma = \sum L_j$: L_j refers to the lateness of a job. It is defined as the difference between the due date and completion time of a job: $L_j = C_j - d_j$. The total lateness, $\sum L_j$, then refers to the sum of each jobs individual lateness.

- $\gamma = \sum T_j$. T_j refers to the tardiness of a job, which like the lateness considers the difference between the due date and completion time. The difference is that if we were to obtain a negative number, corresponding to a job that was completed before its due date, we simply take it as 0. So we still penalize jobs that are completed late, but no longer care how early jobs are completed. Just that they are on time. Mathematically this is defined as follows: $T_j = \max(0, C_j - d_j)$. The total tardiness, $\sum T_j$, is simply the sum of each jobs individual tardiness.

In addition to the regular versions of the objective functions mentioned above, we can also introduce weights w_j to each job, quantifying how 'important' that job is, and take the weighted total completion time or weighted total tardiness for example. Additionally, we could introduce additional costs to certain choices in the schedule and add a whole layer of complexity to the problem. This can be in the form of actual monetary costs associated with certain decisions, but also abstract costs that penalize 'undesirable' situations [12]. For example, in our restaurant we could take the total tardiness of the dishes as our objective, where the due date is a fixed amount of time after the time of taking the order, but we

add a penalty based on how much time apart everyone at the same table gets their food served. This ensures we take individual serving times into account, but still try to serve people at the same table at roughly the same time.

We can make the objective function as simple or complicated as we like, taking all kinds of factors into account. It is important that we strike a balance in our model creation such that the model is complicated enough to accurately reflect reality for us to gain practical results, and also simple enough that existing computational methods can give us results in a reasonable amount of time. For a simple prototype model it may suffice to just pick one of the standard common objective functions, followed by more refinements in a later model when necessary.

4.2 Case Study

In order to decide what objective function we should use, we have to consider what our optimization goal is. Our manufacturer is a contract manufacturing organization (CMO), meaning they provide manufacturing services to other companies. These client companies place an order at our manufacturer, for a certain amount of drugs to be made at a certain price. Once these drugs are finished, the client decides where and how they sell the finished products. This means that our organization simply gets orders from clients, of which prices and conditions can be negotiated beforehand, but when the order is finalized we know exactly what we are dealing with.

For the scope of this research, we take the contract negotiations for granted, and assume placed orders are fixed and unchangeable. Since our prices and production numbers are fixed, we do not need to consider potential profits, aside from certain monetary costs associated with scheduling decisions. After discussion with stakeholders, the scheduling associated costs were considered to be minor enough to be disregarded for the first stages of the model. The main targets that were decided upon were two-fold. Firstly, we would like optimal utilization of the available equipment, as any time that a machine is idle, is wasted capacity. Secondly, as we value our customers highly, we would like to keep customer satisfaction high by limiting any late deliveries.

As mentioned earlier, we need to be careful that our chosen objective function might have undesirable side effects. Say for our first goal we decide that we want to prevent machine idle time by penalizing it in the objective function. As a result some products may be scheduled in such a way that the routing they take involves slower machines, just so the machines are less often idle. According to our objective function, we would have increased the utilization of our equipment, but in reality we just let a product take a sub-optimal routing to seem more efficient. The other goal, limiting late deliveries, can be accomplished by taking either total lateness or total tardiness as our objective function, depending on if we value early deliveries. The added benefit of early deliveries was considered to be minor at best, and thus we decided upon total tardiness as our objective function. Note that making sure all deliveries happen on time as much as

possible, automatically results in efficient utilization of equipment.

Since we want to be able to differentiate between high and low priority jobs, we include the weight component in our objective function. We thus have $\gamma = \sum w_j T_j$ as our objective function, giving us the following three-field notation for our scheduling problem:

$$F \mid r_j, S_{ij} \mid \sum w_j T_j$$

At this point we have our scheduling problem and can choose to implement it and start looking for a solution method, or we can already think about possible refinements we could apply to our model. As we have mentioned before, even though the story here is written linearly, the modeling process is often cyclical. Making additional refinements can be a waste of time if our current model already accomplishes all our goals. During the actual modeling process at the manufacturer the current model was implemented and tested with a solution method. The resulting model with its details can be found in Appendix A. In our case, we unfortunately encountered multiple problems that required some refinements. These problems can be split into two main parts, the first of which has to do with certain practical constraints we had not yet incorporated in our model. These will be discussed in the remainder of this chapter. The second problem we encountered had to do with the size of our problem, making certain solution methods very ineffective. This problem will be addressed when formulating the next iteration of our model, at the end of this chapter. When solving your own scheduling problem, you could decide to skip to chapter 5 for the first iteration of your model and if you run in to similar problems, you can always return here.

4.3 Refinements

Whilst the basic model we arrived at at the end of last section can quickly paint a broad picture of the general form of solutions, and in some cases give sufficiently accurate solutions, in our case most solutions generated were unrealistic or even infeasible in practice. This is well within the line of expectations, as the model was purposefully kept simple, ignoring some practical constraints that are not regularly encountered in existing literature, as they are very problem-specific. For our problem this consists of the following constraints.

- (Constrained cleaning availability) Within some production stages, only a limited number of machines can receive a wet cleaning simultaneously.
- (Expiration dates) The time spent between the end of one stage and the start of the next can be limited by expiration dates, leading to discarded products if these dates are exceeded.
- (Mandatory monthly cleaning) Every machine must receive a wet cleaning at least once every 30 days.

In order to model these constraints effectively, a number of considerations should be taken into account. Given some implementation of one of these constraints, does the increased accuracy of the model weigh up against the loss of computation speed? What effect does the implementation have on the problem's solution space? This last question can be very important for certain solution methods, such as local search methods and derivatives. We return to this when discussing these solution methods.

In the rest of this chapter we introduce a very basic framework, inspired by a method used in drug manufacturing, in order to classify these constraints according to a number of characteristics to help us in finding the most sensible way to model these. Each of these constraints will then be discussed in detail and classified according to the framework.

4.4 Constraint Framework

The framework we use for classifying the constraints is inspired by a framework in drug manufacturing, where deviations in a batch of produced tablets (e.g. dots on the tablets or minor deformities) are classified by considering the gravity, occurrence rate and likelihood of detection of the deviation and depending on these factors, measures are taken to address the situation. Similarly, we can classify constraints by considering the gravity (i.e. how 'bad' it would be if the constraint is breached), the occurrence rate (i.e. how often the constraint would be breached when not implemented in the model), and the likelihood of actual occurrence (i.e. the likelihood that, when the constraint is breached in the model, the constraint is breached in practice). This last characteristic can be very important in scheduling models with uncertainties, as the model becomes less and less reliable when forecasting further in advance. We now classify the previously mentioned added constraints using the framework we created.

4.4.1 Constrained Cleaning Availability

As a result of limited qualified staff, only a certain number of machines within a stage can undergo a wet cleaning simultaneously. One example would be a situation with 4 machines in one stage, where only 2 can receive a wet cleaning simultaneously.

- Gravity: medium. While it is definitely not a desirable situation when this constraint is breached, some careful planning in advance regarding staff availability can make the problem manageable.
- Occurrence rate: high. With the large number of different orders and large amount of time spent when a machine receives a wet cleaning, the odds of two of these cleanings overlapping is very large.
- Likelihood of actual occurrence: low-medium. When scheduling a large number of jobs, reaching well into the future, the likelihood of the exact times being forecast by a solution to be accurate, become increasingly

unlikely. Small deviations in the processing and cleaning times of each process add up over time. This, in combination with unforeseen disturbances due to machine breakdowns and other circumstances, leads to larger deviations further into future. Overlapping cleaning procedures can thus easily move apart, removing the effect of the breached constraint altogether. Conversely, it is also likely that some non-overlapping cleaning procedures end up overlapping in practice.

4.4.2 Temporary Validation

When a product has finished the granulation stage, it is only validated for 1 full month before it should be pressed into tablets, or it hits its expiration date and has to be discarded.

- Gravity: high. Discarding a full batch of a product results in high costs. It also significantly impacts the completion time of the corresponding order, thus having a large effect on our main objective.
- Occurrence rate: low. Generally speaking, the time spent between granulation and tablet pressing is well below this 1 month time window. This is largely due to the production capacity in the tablet pressing stage being slightly higher than the capacity of the granulation stage, resulting in a slight bottleneck.
- Likelihood of actual occurrence: high. When this constraint is breached, the times at which the affected product is scheduled for granulation and tablet pressing are separated by at least 30 days, but potentially more. While it is possible for this time to be very close to 30 days and the same disturbances as described for the previous constraint to reduce this separation, we can assume this does not happen often.

4.4.3 Periodic Cleaning

Every machine must receive a wet cleaning at least once every 30 days.

- Gravity: high. When a machine has not received a wet cleaning for 30 days, it cannot be used at all, leading to large delays.
- Occurrence rate: medium. For most stages it is an unlikely occurrence, as it would require only one type of product to be processed for 30 days. When dealing with changing products a wet cleaning would be required regularly either way, resulting in the 30 day mark never being reached. For a few specific machines that are mostly catered to a specific product, this may occur every now and then.
- Likelihood of actual occurrence: high. This is as a result of the same argument as that of the previous constraint.

Note that if we were to use the framework to any of the constraints present in the basic model, we would find their gravity and occurrence rate to be very high, as a breach would result in practical infeasibility, showcasing that these constraints are indeed important to be modeled

4.5 Constraint Handling

There are multiple commonly used methods in scheduling theory to deal with these additional constraints [13]. Below, we discuss three of these methods and their pros and cons: hard constraints, penalty-based constraints, and post-processing repair methods.

4.5.1 Hard Constraints

One direct way to handle constraints is by hard implementation, strictly enforcing the constraint. This can be achieved in multiple ways, for example by disregarding infeasible solutions altogether, or by incorporating the constraint in certain solution methods, making sure they never find such an infeasible solution.

Pros:

- Guarantees feasibility, we never end up with a solution that does not work in practice.
- Can significantly reduce the number of valid solutions we need to consider, potentially improving efficiency in finding the best solution.
- Very easy to implement for certain solution methods.

Cons:

- Can overly restrict potential solutions, eliminating promising solutions that only slightly violate constraints. This can be especially detrimental in certain solution methods such as local search methods.
- Implementing multiple hard-coded constraints may make finding valid solutions very difficult, leading to a loss of efficiency in finding the best solution.

4.5.2 Penalty-Based Constraints

Instead of strictly enforcing constraints, an alternative is to allow violations but assign penalty points whenever constraints are breached. These penalty points can then be incorporated in the objective function, scoring solutions that violate these constraints as worse, but not outright forbidding them. A useful extension of this approach is to let the penalty points scale dynamically with time. This reflects growing uncertainty, especially in scheduling problems that schedule far

into the future.

Pros:

- Allows for a trade-off between feasibility and quality of solutions.
- Works very effectively for certain solution methods, such as most heuristic and metaheuristic approaches.
- Allows for dynamic scaling of constraints.

Cons:

- This approach requires careful consideration of the value of penalty weights. If they are too low the constraint may be ignored, but if they are too high, the solution space may become too restrictive
- There is no guarantee of obtaining a fully feasible solution at the end.
- May be computationally expensive, depending on how complex the penalty calculations are.

4.5.3 Post-Processing Repair Methods

The third approach we are considering is simply ignoring the constraints initially. After the solution method has obtained a final solution we apply a repairing algorithm to adjust the solution in such a way that we obtain a feasible solution.

Pros:

- Allows solution methods to explore a wider solution space, potentially finding better solutions
- Usually computationally faster than the other methods, with the chance of obtaining a feasible solution at the end

Cons:

- Reparation of the final solution may be both expensive and result in a bad solution.
- There is no guarantee that the final solution is repairable at all.

4.6 Refined Model

Having seen some potential implementation methods for our additional constraints, we would like to put them to the test. Constraints with a high gravity if broken should probably either be hard-implemented or repaired during post-processing. Using penalty-based constraints would require a large penalty, having potential bad effects on the solution space. Due to the low occurrence

rate of the Temporary Validation constraint, we might be able to get away with using reparation methods. For the Periodic Cleaning constraint, having high gravity, as well as a medium occurrence rate, we try solving this by hard implementation. Lastly, the Constrained Cleaning Availability has some flexibility, making it more of an undesirable situation than a hard constraint. This is perfect for penalty-based constraints, which is what we are going to use for this constraint. In consultation with the company of our case study, we decided a fitting penalty for this constraint would be the equivalent of a single product being a week late, or 5 business days. The penalty value is thus set to be equal to 5.

Making these decisions is no hard science and at the end of the day some of our choices may be sub-optimal. This is no big problem though, as we could always come back to this step and reevaluate our options if our results are lacking or run into some quirks we had not anticipated.

Before we move to a refined model formulation, it is important to know what solution method we wish to use, as the way we define our problem may depend significantly on this decision. Which is why in the next chapter, we take a look at the different solution methods available to us.

Chapter 5

Solution Methods

Before we can finalize our model, we want to take our intended solution method into consideration, as it may have an effect on previously made choices. Therefore, we now provide a brief overview of commonly used solution methods for mathematical modeling. We can generally divide these solution methods in two categories. Firstly, there are exact methods, which at the cost of lengthy calculations, provide the absolute best possible solution. The other main category is heuristic approaches, which are usually considerably faster than exact methods, but only approximate the optimal solution, being unable to guarantee the best solution is found. In the remainder of this section, we consider both of these categories in more detail, discussing their main concepts and commonly used methods that fall in these categories.

5.1 Exact methods

Naturally, an ideal solution method would be one that finds the exact optimal solution. In order for such a method to work, it would have to both find the optimal solution as well as prove that it is indeed optimal. In the case of scheduling, where usually only a finite number of potential solutions exist, this could be done by exhausting all possible solutions, and thus, showing the best one is optimal. Unfortunately, this is almost never feasible in practice. Even for a small schedule, where only 20 jobs need to be sequenced, the number of possible sequence is reaching into the quintillions ($20! \approx 2.43 \cdot 10^{18}$). Even if we had a computer capable of evaluating a billion sequences every second, it would take over 70 years to be finished.

Luckily, our options are not limited to brute force. But before we can move on to other solution methods, we take a look at a common way of describing mathematical problems. This method consists of a whole family of methods, known as mathematical programming [14]. Some well-known members of this family are linear programming, integer programming and constraint programming. In general, mathematical programming is used to define a problem as a

combination of an objective function and one or more constraint functions, all of which are functions dependent on some fixed parameters and variable input, known as decision variables. The model formulation we have already created (in appendix A), follows this system. Depending on the exact variant of programming, there may be some additional rules regarding the constraint functions and/or decision variables. For example, in integer programming all decision variables are required to be integer and in linear programming all constraints and the objective function must be linear functions.

To illustrate these concepts, we take a look at a concrete example. For this example, we use integer linear programming (ILP), in which all decision variables are integral, and the constraints and objective functions are linear. Consider the following ILP.

$$
\begin{aligned}
&\max x + 2y \\
&s.t. \\
&3x + 2y \leq 12 \\
&2x + 3y \leq 12 \\
&-x + y \leq 1 \\
&x, y \geq 0 \\
&x, y \in \mathbf{Z}
\end{aligned}
$$

Once we have our formulation, the goal is to find values for x and y such that the objective value is as large as possible, while all constraints remain valid. We first introduce some commonly used terminology for clarity. A *solution* is an assignment of values to decision variables. $(x, y) = (1, 1)$, $(x, y) = (2, 2)$ and $(x, y) = (10, 0)$ are all examples of solutions. A *feasible solution*, is a solution for which all constraints are valid. For example, $(x, y) = (1, 1)$ is a feasible solution, since all inequalities hold and x and y are nonnegative integers. An *infeasible solution* is one that does not pass all constraints. $(x, y) = (10, 0)$ is an infeasible solution, since $3x + 2y = 3 \cdot 10 + 2 \cdot 0 = 30 \not\leq 12$. We call the best possible objective value the *optimal value* and any solution that has the optimal value as its objective value, is an *optimal solution*. Note that as mentioned before, there may be more than one optimal solution, but only one optimal value can exist.

The above problem can be solved very fast, due to its small size. Both brute force and visual methods work well here, but neither are feasible for large-scale problems. In the remainder of this section we discuss the Branch and Bound algorithm, which finds a lot of use in solving ILPs and other mathematical programming problems.

5.1.1 Branch-and-Bound

Branch-and-Bound (B&B) is an algorithm that recursively divides our problem into smaller sub-problems (or *branches*) and solves the main problem efficiently

by 'ignoring' large amounts of these branches which is made possible through keeping track of certain bounds [15]. This way, we are essentially conducting an exhaustive search, but in a way faster manner than if we were to check every possible solution individually. Before we explain the bounding process, we take a look at an example using just branching methods.

Consider the ILP from before. We can see that x can never be larger than 4, otherwise the first constraint would be broken, as $2y$ can never be negative. Similarly, we find that y can never be larger than 4 due to the second constraint. We thus have 5 possible values for both x and y, as neither can be negative. We can now branch our problem into two subproblems by restricting each subproblem to certain values of x and y. There are lots of options regarding how we branch exactly, but for the purpose of this example, we choose an option where we encounter some nice consequences. As mentioned before, y cannot be negative or larger than four. Out of the remaining 5 options, we try to split it somewhat evenly by having one branch with $y \geq 3$, and the other branch with $y \leq 2$.

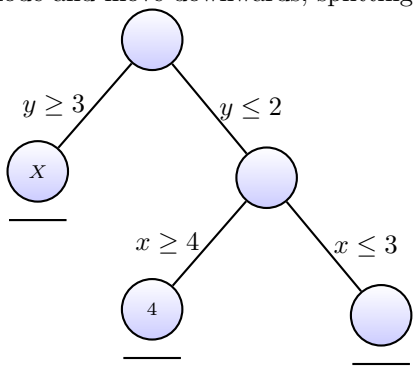
- $y \geq 3$: Since y is at least equal to 3, we find that x can at most be 1, due to the second constraint. But because of the third constraint we can see that x needs to be at least 2. Seeing as we have arrived at a contradiction, there are no feasible solutions in this branch, meaning we can disregard it entirely.

- $y \leq 2$: We can try to reason our way to a solution like above, but we can also decide to branch further. Let us say we now branch on the x variable, with our two options for example being $x \geq 4$ and $x \leq 3$.

  - $x \geq 4$: Since x is at least equal to 4, we find that y is at most 0, due to the first constraint. We also know that x cannot be larger than 4 due to the first constraint, so this branch has a singular solution $(x, y) = (4, 0)$ with objective value 4.
  - $x \leq 3$: At this point we can branch at another point, perhaps further restricting x, or returning to restrictions on y.

If we continue this branching method we would eventually consider all feasible solutions and thus find the best one. In this way, we can prevent checking a lot of possible solutions that turn out to be infeasible, but we still consider all feasible solutions, which can still be an enormous amount. We thus would like to be able to skip some solutions that are feasible, but not optimal. One way to do this is by keeping track of any feasible solutions we have already found and checking these against the theoretically best possible solution a branch could give. For example, if we found a solution with objective value 6 at the end of a branch, it would be pointless to check another branch of which we know that there can be no solution with objective value greater than 4. What we just described is essentially the bounding process. We keep track of solutions we have already found (a lower bound on the best solution, since the best solution is at least as good as this one) and in a specific branch we keep track of how good

Figure 5.1: A graph showcasing the branching process. We start at the top node and move downwards, splitting into two paths each time we branch.



solutions can be in the best case scenario (an upper bound in the branch). If the upper bound in a branch is lower than the lower bound of the entire problem, we can get rid of, or *prune*, the branch. This sounds great, but we still need a method to actually find out how good solutions can be in a certain branch. This is where relaxations come in.

A relaxation of an ILP is created by removing one or more of the constraints of the problem. When we remove a constraint, any previous feasible solution is naturally still feasible, as all other constraints still hold. There is a possibility that some solutions that used to be infeasible have now become feasible, if the infeasibility was only caused by the removed constraint. As a consequence, the optimal value of the relaxed problem can be equal to or better than the optimal value of the original problem. Solving the relaxed problem thus gives us a upper bound on the optimal value of the original problem, which is what we wanted for our Branch-and-Bound algorithm.

An example of a relaxation would be in our ILP to disregard the second and third constraints. The best possible solution in that case would be $(x, y) = (0, 6)$, with an objective value of 12. If we use the same relaxation in the branch where $y \leq 2$, we find an upper bound of 6, where $(x, y) = (2, 2)$. This particular solution also happens to be feasible in the original problem, with all constraints. We have thus found a global lower bound of 6 and a local upper bound of 6 in this particular branch. Combining this with the fact that we already pruned $y \geq 3$, we have actually found the optimal solution and proven its optimality.

There are some important considerations for how we pick our relaxation. Ease of computation is essential, since if the relaxed problem is not easily computable, it would hardly reduce the effort we need to make. We also require our relaxations to be not too relaxed, since removing too many constraints could give us unhelpful information. If we had considered the relaxation of removing constraints 1 and 2, our problem would become unbounded, as taking $(x, y) = (1000, 1000)$ would give a feasible solution with objective value 3000, and we can keep on increasing the numbers to infinity. The most commonly

used relaxation tackles the first consideration very well and is often times also helpful for the second. This relaxation is also named the *natural relaxation* and consists of dropping the integrality constraint, allowing the decision variables to take decimal values. Without this restriction, there are methods for computers to solve these problems incredibly fast. Whenever we branch our problem, we can simply use the natural relaxation to obtain a local upper bound for the branch, and when we at some point encounter a globally feasible solution we obtained a global lower bound.

There exist other variations of this algorithm, for example Branch-and-Cut, making use of slightly different methods, but the overall approach is similar.

5.2 Heuristics

When exact solutions are either impossible to obtain or the process takes too long to be useful, heuristic methods are the next best thing. These methods tend to be way faster, at the cost of being unable to guarantee an optimal solution will be found. Even when an optimal solution would be found, the method is not able to prove this optimality. Many of these methods have been tested thoroughly and produce great results in practice. Two families of heuristics methods will be discussed in the remainder of this section: Constructive heuristics and metaheuristics

5.2.1 Constructive Heuristics

Constructive heuristics are algorithms that are designed to construct a feasible solution. Many of these heuristics are made specifically tailored to a single problem, making use of some property of the problem definition, the input data, or a combination, such as apparent bottlenecks. These algorithms can range from simple dispatching rules, to very complicated processes. Some very effective constructive heuristics have been developed for production planning. Unfortunately, these tend to be limited either to basic models, being unable to cope with the added constraints encountered in practice, or are very specific, being unable to adapt to slightly different situations. Simple dispatching rules however, are capable of dealing with any kind of constraints and are a great way to compare other methods to a "naive" human approach, such as scheduling jobs in the order they should be finished.

When dealing with relatively simple or common problems, there is a good chance someone already came up with a great constructive heuristic, or some of the simpler dispatch-rule based heuristics can give good solutions [16]. When dealing with more complicated, novel problems, using existing constructive heuristics may have varying results, while coming up with a new tailored heuristic is a daunting task.

5.2.2 Metaheuristics

Whereas constructive heuristics tend to be either very simple or very specifically tailored to a particular problem, metaheuristics form more of a general approach to solving optimization problems. The process of a metaheuristic starts with some feasible solution, usually obtained by a constructive heuristic, and then through a combination of searching for better solutions and refining found solutions tries to jump from solution to solution until some predetermined condition, such as a certain amount of time having elapsed, has been met.

To better understand this iterative process, it may prove helpful to introduce the concept of a *solution space*. The solution space represents the set of all possible solutions to the problem at hand. Metaheuristics move from one solution to another solution by navigating this solution space.

There are several types of metaheuristics, working in different ways [17]. Some are more exploration focused, searching many areas of the solution space in hopes of finding good areas. Some are more exploitation focused, sticking close to previous solutions, refining them until no more simple improvements can be made. In order to gain an idea how these methods work, we enter a metaphor describing local search metaheuristics.

5.2.3 A Mountainous Walk

Suppose you find yourself in a mountainous landscape, and you wish to travel to the highest point. How do you find it? Simply looking around you and spotting the highest point seems like a simple solution, but there is a slight issue, namely that there is an incredibly thick mist all around you, rendering your vision useless beyond a couple of meters. Luckily you are equipped with an altimeter, capable of accurately measuring your current altitude.

A logical next approach would be to stand at your starting point, wander a bit around that point, noting down the altitudes in a couple meters radius and picking the highest point in that area. This will be our new starting point, from which we repeat this procedure, every time moving towards the highest point in our area, until we arrive at some sort of peak. We have definitely made some improvements from our original starting point, and there is no immediately obvious way to improve. We may even be at the absolute highest point in our landscape. It is more likely however, that we are simply on top of one of the many smaller hills and peaks dotted around this area, missing the actual high mountains. We can never know for sure though, unless we find a higher place, as we have no idea what the altitude of the highest point even is.

One way to circumvent getting stuck on a small hill, is by bringing a bunch of flags with us, that we stick into the ground where we have been. Whenever we are deciding our new starting point, we disregard any points where a flag is already planted, making sure we do not visit the same place twice. On a small, steep hill we might place a couple of flags and go on our merry way to the next peaks, but when we are at a less steep, more wide hill, such as a raised plateau, we might need to place a ludicrous amount of flags before we can finally move

on to a new area. The general composition of the larger area has great impact on how well our searching method performs.

5.2.4 Returning to Math

What we described above is essentially how local search metaheuristics work. The mountainous landscape is our solution space, and the highest peak is the optimal value we are searching for. The thick mist and altimeter represents us only being able to see the height of places where we actually calculate the objective value. Unlike in our example, where we are dealing with a nice 3-dimensional area we need to search through, the solution spaces we need to traverse in these optimization problems tend to have hundreds or more dimensions. Another, less important difference is that we are oftentimes searching for the lowest point instead of the highest, when minimizing something. This is just a difference in perspective though, and has no influence on our solution method.

In our metaphor, we described local search methods, where the process of going from one solution to another consists of small steps. Scheduling-wise this can be thought of as a minor or singular change to our schedule. Say we have a sequence of 100 jobs and we pick 2 out of them and swap their places, that would be an example of a small change. In addition to local search methods, there are also metaheuristics based on genetics, fittingly called genetic algorithms, where a group of solutions is affected by selection, crossover, and mutation principles. Many more metaheuristics exist, oftentimes inspired by some natural phenomenon, each of them with their own pros and cons. To limit our scope, we are sticking to local search methods in this section, as they provide some of the more intuitive and easily implementable options, whilst still giving enough variation for varying problems and solution spaces.

5.2.5 Examples of Local Search Methods

During our mountainous walk we have already encountered two local search methods: hill climbing and tabu search. In addition to those there are many more options and many variations of each of them. We now consider a few of them in more detail.

- Hill climbing: Hill climbing is a simple iterative optimization approach that starts with some initial solution and incrementally improves upon itself by making small changes. The process continues until no further improvements are possible. This can be compared to simply walking to the highest near point repeatedly, until we are on top of some peak. Unfortunately, this is not necessarily the highest peak in the mountainous landscape. It could very well be the peak of a small hill that stands next to a large mountain. So, while this method is very simple to implement, and is computationally efficient, it may easily get stuck on some suboptimal peak. In mathematical terms we call this suboptimal peak a *local*

optimum. This simply means that in the immediate surroundings, or locally, this is is the best solution, but it does not have to be the best overall solution, which is similarly referred to as the *global optimum*. This method is often combined with other methods, such as generating a bunch of random starting points that each use hill climbing, to get a varied idea of peaks around the solution space.

- Tabu search: Tabu search functions almost identically to hill climbing, with the only extra addition being that we keep track of a list of points we have already visited, called the tabu list. This is done to avoid revisiting places we have already been. This way we can escape local peaks. For large problems the tabu list can grow very large, so careful consideration is required on how we deal with this list. Do we remove points we have not visited in a long time to save memory? Do we only put some points, that we deem likely for revisiting otherwise, in our list?

- Simulated annealing: Unlike with the previous two examples that were completely deterministic, simulated annealing introduces a probabilistic factor into the mix. The method is inspired by the annealing process in metallurgy, where materials are heated up before being slowly cooled down, removing defects. Like before, we start with some initial solution, but now instead of computing many close solutions, we only generate a single one. If this generated neighbor is better than our current solution, we accept it, taking it as our new current solution. If the solution is worse, we do not immediately discard it. Instead, we accept it with some probability, and discard it otherwise. This probability depends both on the difference between objective values (solutions that are much worse than the current solution are less likely to be accepted), and the current 'temperature', which is an artificial parameter that decreases in value over time (earlier on in the process we are more likely to accept worse solutions than later). Due to the probabilistic nature, this method is capable of escaping local optima very well and is in general applicable to a wide range of problems. The method does require some additional tuning with regards to the exact probabilities, 'cooling process', and method of generating neighboring solutions.

5.3 Case Study

Picking a suitable candidate solution method for our case study requires a couple of considerations. First of all, we can question how the existing exact methods and heuristics compare. The size of the problem plays a big role in this, so we start by exploring that.

5.3.1 Problem Size

For the application we are trying to solve, we can regularly expect problem instances where we need to sequence around 125 product orders. The number of different machine assignments a product order may have varies per product, but on average this is at least 6. The sequence of products may vary per stage, allowing for even more possibilities, but if we disregard this option for now, we would already find ourselves with $125! \cdot 6^{125} \approx 3.50 \cdot 10^{306}$ different possible schedules, which is simply a ludicrously large number. While this is not an immediate problem for exact methods, which could make use of 'nice' structures and patterns in the problem, it turned out in our case that the use of state of the art exact methods was far too slow. We thus have to settle for heuristics.

5.3.2 Heuristics

Seeing as our specific problem is slightly different as commonly discussed problems in literature, we are limited in simply using strong existing constructive heuristics. Since creating our own constructive heuristic or adapting an existing one to our problem is no trivial task, a good starting point is by sticking to more easily implementable options, such as dispatching rules and local search based metaheuristics. Since local search methods start from some initial solution, preferably a good one, our main approach is as follows. We start by generating a decent initial solution using dispatching rules, followed by refining this using local search methods.

Currently, both simulated annealing and tabu search seem like good options for our solution method. A more detailed overview of both of them can be found in Appendix B and Appendix C respectively. In order to choose between these two options, we test both of them on a real problem instance from our case study. Since the performance of both methods depend on a combination of parameters and options, we test the solution methods in combination with different options regarding our parameters. This process can be found in Chapter 6.

Chapter 6

Tuning

Now that we have properly defined a mathematical model describing our problem, being able to deal with the constraints we encountered in practice, as well as having decided on two candidate solution methods for our problem, we can finalize our model by picking one of these solution methods, and tuning the details in our approach. To this end, there are three areas in which we may want to tweak our approach. Firstly, we want to finalize our solution method. Next, we have a look at options regarding a solution space reduction. Lastly, we would like to test our choices regarding constraint handling.

All tests performed in this chapter have been carried out by the same personal computer, using a script written in the programming language Python (version 3.11.5). The specifications of the computer are as follows:

- CPU: AMD Ryzen 7 5800X3D 8-Core Processor.
- Storage: 1TB SSD.
- Memory: 32GB.
- Operating System: 64-bit.

The tests themselves consist of trying to find the best possible solution to a practical problem instance based on real data from our case study. Each of the tests has been performed at least 5 times to reduce the effect of randomness associated with our solution methods. The remainder of this chapter goes into further detail on the three areas in which we may want to tweak our approach.

6.1 Simulated Annealing or Tabu Search

Aside from picking between simulated annealing and tabu search, two main factors in how effective either method is, are the choice in neighbor generation and the precise tuning of method-specific parameters. In simulated annealing this parameter is the temperature, and in tabu search we want to see the effect of the number of neighboring solutions we generate at each step.

6.1.1 Neighbor Generation

For neighbor generation, we consider the following methods:

- Single job reassignment: Pick a job and stage, and reassign this job to another eligible machine in that stage.
- Single sequence swap: Pick 2 jobs in the same stage in our schedule and swap their position in the sequence for that stage. See Figure 6.1.
- Single sequence insertion: Pick a job and stage, and insert this job at some other point in the sequence for that stage. See Figure 6.2.

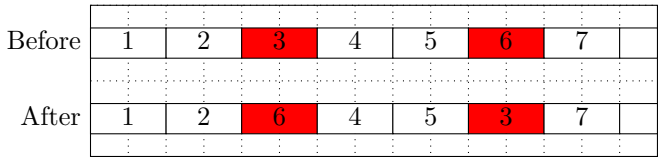


Figure 6.1: Here you can see an example of a sequence swap. Here we swap the two jobs marked in red, 3 and 6. They simply swap places in the sequence without affecting the order of the other jobs.

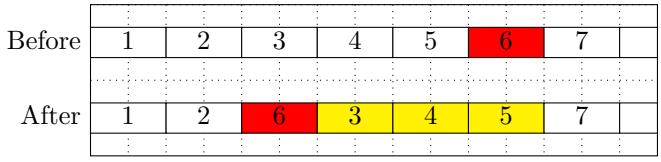


Figure 6.2: Here you can see an example of a sequence insertion. Here we insert job 6 into position 3 of the sequence. In order to make room for job 6, the other jobs shift over one place, as outlined in yellow.

Since we want our solution method to be able to change both the job sequence and job assignment when creating neighbors, we definitely include job reassignment into our strategy. We can then choose between the two options for alterations to the sequence: swapping and insertion.

Both of these are easy to implement and have their own benefits and drawbacks. Swapping is good for making small, incremental changes to an already generally good solution, but may struggle a bit more to significantly alter a solution, meaning it struggles with deep local optima. Insertion on the other hand introduces some larger changes to the overall solution in each step and is better suited for escaping local optima, but can also work a bit too disruptive and 'waste' promising partial sequences.

Instead of using one or the other we could also use a combination of them, either deterministically taking turns or probabilistically picking which of the two

is used. More complicated strategies may also prove useful, preferring insertion at the start of the search to do the rough work and when promising solutions are found we can finetune with swaps. In our case we stick with the following three options:

- Purely swapping.
- Purely insertion.
- Use both, probabilistically with each having a 50% chance of occurring.

6.1.2 Method-Specific Parameters

When using simulated annealing and determining the probability of accepting a neighbor as the new solution, we consider the factor of difference in objective value, divided by current temperature. Choosing an initial temperature of 1 or 1000 has a profound effect on how likely we are to accept worse solutions during our process. Setting T too low means we only consider slightly worse solutions, which in turn makes us more susceptible to local optima again. On the other hand, if we take T too high, we risk accepting unnecessarily bad solutions during our search, significantly slowing down our process. Since it is hard to estimate or justify what a good initial temperature would be, we test a few different options:

- T = 1
- T = 5
- T = 10
- T = 25

When using tabu search, the number of neighboring solutions we generate at each step, can have a profound effect on the effectiveness of our model. If we generate too few solutions, the average quality of our accepted solutions decreases, potentially decreasing the quality of our overall search. On the other hand, if we generate too many solutions, the computational cost of each iteration can get very high, leading to a significantly slower search. In our tests we try the following options:

- Number of neighbors generated = 5
- Number of neighbors generated = 10
- Number of neighbors generated = 20
- Number of neighbors generated = 50

6.1.3 Results

For the tests we run in this section, we used a real problem instance from the company of our case study that consists of 98 job orders. For all of these tests the maximum runtime was set to 5 minutes.

Neighbor Generation: Purely Swapping

In Table 6.1 and 6.2, we can see the results when using the purely swapping method for neighbor generation. The numbers correspond to the final objective value, which is the total tardiness of all jobs together with an extra 5 for each time the constrained cleaning availability constraint is breached. Note that since we are minimizing, lower objective values are better.

Table 6.1: Overview of objective values obtained when using tabu search and only sequence swapping. Each column corresponds to the number of neighboring solutions generated at each iteration of the algorithm. These are respectively 5, 10, 20, and 50.

Test	5	10	20	50
1	752.19	498.50	447.63	444.85
2	780.11	466.84	444.41	441.11
3	674.46	494.49	449.32	445.01
4	802.68	475.19	449.39	445.11
5	763.13	511.65	443.35	445.61
average	754.51	489.34	446.82	444.34

Table 6.2: Overview of objective values obtained when using simulated annealing and only sequence swapping. Each column corresponds to the initial temperature used in the algorithm. These are respectively 1, 5, 10, and 25

Test	T=1	T=5	T=10	T=25
1	440.98	440.70	440.98	440.98
2	440.70	441.19	440.70	440.70
3	440.98	440.70	440.70	440.70
4	441.02	440.98	442.74	441.02
5	440.98	441.25	440.98	440.98
average	440.93	440.96	441.22	440.88

Neighbor Generation: Purely Insertion

In Table 6.3 and 6.4, we can see the results when using the purely insertion method for neighbor generation.

Table 6.3: Overview of objective values obtained when using tabu search and only sequence insertion. Each column corresponds to the number of neighboring solutions generated at each iteration of the algorithm. These are respectively 5, 10, 20, and 50.

Test	5	10	20	50
1	558.52	452.33	441.35	444.61
2	592.14	459.35	440.71	445.01
3	589.20	449.45	441.84	440.70
4	586.03	456.58	443.65	445.01
5	589.71	463.09	441.47	441.02
average	583.12	456.16	441.80	443.27

Table 6.4: Overview of objective values obtained when using simulated annealing and only sequence insertion. Each column corresponds to the initial temperature used in the algorithm. These are respectively 1, 5, 10, and 25

Test	T=1	T=5	T=10	T=25
1	440.70	440.70	441.02	440.98
2	440.70	440.98	440.98	441.02
3	440.70	440.70	440.98	441.61
4	440.70	440.70	440.98	441.28
5	440.70	440.70	440.70	441.02
average	440.70	440.76	440.93	441.18

Neighbor Generation: Probabilistic Mix

Lastly, in Table 6.5 and 6.6, we can see the results when using the probabilistic combination of both methods.

Table 6.5: Overview of objective values obtained when using tabu search and both sequence swapping and insertion. Each column corresponds to the number of neighboring solutions generated at each iteration of the algorithm. These are respectively 5, 10, 20, and 50.

Test	5	10	20	50
1	679.14	466.94	441.73	440.98
2	667.25	464.03	442.66	441.02
3	654.33	466.27	442.33	440.70
4	571.35	464.06	441.41	440.70
5	596.00	472.18	442.90	440.85
average	633.61	466.70	442.21	440.85

Table 6.6: Overview of objective values obtained when using simulated annealing and both sequence swapping and insertion. Each column corresponds to the initial temperature used in the algorithm. These are respectively 1, 5, 10, and 25

Test	T=1	T=5	T=10	T=25
1	440.70	440.98	440.98	444.72
2	440.70	440.70	441.02	440.98
3	440.70	440.70	441.02	445.01
4	440.70	440.98	440.98	445.01
5	440.70	441.02	442.13	441.02
average	440.70	440.88	441.23	443.35

6.1.4 Conclusion

Based on the results we have seen in this section, we can make a couple of observations. When looking at the results for tabu search, we see that the algorithm performs significantly better when the number of neighbors we generate at each iteration is 20 or 50, compared to 5 or 10. We also observe some differences between the different types of neighbor generation, but these differences get smaller the more neighbors we generate. For simulated annealing we see less varied results. For this particular problem instance, the algorithm seems to perform very similar regardless of the initial temperature and neighbor generation method we choose.

Since both methods are capable of generating equally good solutions at this point with certain parameter setups, we do not see a clear winner among them. At this point we could either continue comparing both methods on other problem instances, or we can simply pick one of the two methods and continue from there.

If we do ever run into major issues, we could always revisit this step to give the other method another try. In our case, we choose for the second option and we pick simulated annealing as our method for the remainder of this report. In order to evaluate the performance of the simulated annealing algorithm over time, we have recorded and plotted the progression of solution quality for some of the above tests. The corresponding graphs can be found in Appendix D.

6.2 Current Model

Now that we have decided what solution method we will use, we can give an overview of our model. Since we are not using any exact methods, and some of the additional constraints are difficult to express in an ILP formulation, we are skipping over that here. Additionally, we do not need an ILP formulation as we are using heuristics to generate each solution. We can describe our method as the following broad step-by-step method:

1. **Parametrization.** Set initial values for parameters associated with simulated annealing. In our case we can start out by setting our initial temperature $T_0 = 5$ and set a maximum runtime $t_{max} = 300$ seconds and change both of these parameters later based on our results. Additionally we introduce variables that keep track of our current solution and the best found solution, including their objective values.

2. **Initialization.** Generate an initial solution heuristically. This is done by simple dispatching rules, sequencing all jobs in the order of their due dates and sticking with that order for each production stage. Using that order, we are assigning them to the machine where they would be able to start the earliest.

3. **Solution Evaluation.** Evaluate the objective value of our initial solution. In order to do this, we make use of an algorithm that can be found in Appendix E. We proceed by setting both the current and best found solution equal to our initial solution.

4. **Neighbor Generation.** Generate a neighboring solution. We can do this using one of the previously used methods, such as sequence swapping and insertion.

5. **Solution Update.** Evaluate the objective value of the neighboring solution and compare with our current solution. If the new solution is better we update it to be our current solution and compare it with the best found solution, updating that as well if necessary. If the new solution is worse, we compute the quantity $P = e^{-\frac{\Delta v}{T}}$ with Δv the difference between the two objective values. We then with probability P choose to update our solution anyway, setting the neighboring solution to be our current solution.

6. **Temperature Update.** We update our temperature as follows: $T = T_0 \cdot \frac{t_{max}-t}{t_{max}}$, where t is the number of seconds that passed since the start of the algorithm.

7. **Check Termination Condition.** Check whether $t \geq t_{max}$. If yes, we end our search and output the best found solution and corresponding objective value. If no, we return to step 4.

6.3 Solution Space Reduction

For simulated annealing and other local search based metaheuristics, the effectiveness of the method is strongly dependent on the diameter of the solution space. The diameter of the solution space is defined as the largest distance between two points in the solution space, where the distance between two points is defined as the number of steps required to take to go from one point to the other. What a step means precisely depends on our neighbor generation method, but for all options we are considering, the diameter is equal. Without loss of generality we can assume for this purpose we use the following neighbor generation methods:

- Single sequence swap: Pick 2 jobs in the same stage in our schedule and swap their position in the sequence for that stage.
- Single job reassignment: Pick a job and stage, and reassign this job to another machine in that stage.

If we suppose a problem instance of 125 jobs and 4 stages, where 2 of the stages have meaningful assignment alternatives for each job, the diameter of our problem can be computed as follows. Going from one solution to any other solution requires at most 124 swaps per stage and 2 reassignments per job, bringing us to a total of $124 \cdot 4 + 125 \cdot 2 = 746$ operations, giving us a diameter of 746.

A smaller diameter means an easier route from any starting point to good solutions, resulting in better efficiency. One consideration we could make is whether we might benefit from removing a large chunk of our solution space, potentially removing the best solutions, in exchange for greatly reduced diameters. In practice we found that between production stages there is oftentimes only slight changes in the job sequence. We could make use of this fact by enforcing each stage adopting the same job sequence, this would reduce our diameter roughly by half. In multi-stage machine environments this is also known as a permutation problem, transforming our flow shop problem into a permutation flow shop problem.

In addition to the options of using a unique sequence for all stages or the same for all of them, we may also want to try an in-between option we refer to as half-permutation, where the first 3 stages share the same sequence, but the packaging stage has its own sequence. Instead of leaving all the sequencing problems to the metaheuristic algorithm, we could also try to reduce our solution

space by using a constructive heuristic to sequence certain stages. We could for example use the metaheuristic to determine the sequence at the first production stage, followed by dispatching rules sequencing jobs as they become ready for the next stage.

In total we tested the following five approaches:

- Full permutation (FP)
- Half permutation (HP)
- Unique sequence per stage (US)
- First stage metaheuristic followed by dispatching rules (FD)
- Half permutation followed by dispatching rules for packaging (HD)

These tests were performed on a slightly larger problem instance than before, consisting of a total of 125 job orders. The maximum runtime was set to 500 seconds for each of these tests. The results can be found in Table 6.7:

Table 6.7: Overview of objective values obtained when comparing different methods of solution space reduction with a runtime of 500 seconds. Each column corresponds to a particular method.

Test	FP	HP	US	FD	HD
1	551.13	577.43	623.67	579.46	584.16
2	559.11	599.21	621.80	593.60	595.03
3	567.30	562.65	608.99	581.58	600.60
4	538.63	601.35	620.05	591.67	589.77
5	544.91	575.83	621.41	569.88	582.08
average	552.22	583.29	619.18	583.24	590.33

We can clearly see the full permutation method performing the best at this runtime of 500 seconds. Even when significantly reducing the runtime to just 50 seconds, we can still see full permutation coming out as the best of the test. See Table 6.8. Note that because of the reduced runtime, we may experience more influence from the randomness in our solution method, and thus decided to double the amount of tests we ran for each option.

Table 6.8: Overview of objective values obtained when comparing different methods of solution space reduction with a runtime of 50 seconds. Each column corresponds to a particular method

Test	FP	HP	US	FD	HD
1	633.17	701.21	646.39	710.26	762.33
2	598.53	817.70	661.99	673.12	827.56
3	597.77	709.53	683.60	690.26	808.70
4	645.99	764.78	660.69	645.43	763.70
5	601.99	817.96	660.47	807.15	718.40
6	649.12	751.50	663.57	651.19	724.37
7	643.71	791.06	664.38	648.04	784.18
8	622.43	763.89	647.32	615.77	722.25
9	635.25	776.24	657.31	679.74	638.35
10	601.90	666.59	662.93	671.82	680.01
average	622.99	756.05	660.87	679.28	742.99

6.4 Constraint Handling Comparison

Having finetuned our solution methods, we would like to revisit chapter 4 and investigate the effects of the additional constraints we added. We do this by testing a couple of hypothetical situations where one or more of these constraints are removed. We cannot just compare the results afterwards and conclude whether adding the constraint was good or not, since the feasibility of certain solutions may change when discarding certain constraints. We can however explore how big of an effect certain constraints have on the quality of solutions.

We are thus testing the following cases:

- Adding in a temporary validation constraint.
- Remove the constrained cleaning availability constraint.
- Our baseline without changes.

For these tests we use the same problem instance that we used when testing solution space reduction methods. We are also using a maximum runtime of 500 seconds. The results can be found in Table 6.9.

Table 6.9: Overview of objective values obtained when comparing different configurations of our constraint handling approach.

Test	Add TV	Remove CC	Baseline
1	550.59	549.57	555.57
2	557.73	549.70	553.99
3	556.51	542.41	583.26
4	542.26	565.66	552.60
5	568.69	537.01	537.23
average	555.16	548.87	556.53

We observe that by adding the temporary validation constraint, we marginally improve our average found solution, which is slightly unexpected. We can thus improve on our solution method by incorporating this constraint. In all our baseline tests, none of the final solutions were infeasible according to the temporary validation constraint. We also observe slight improvements in our solution when disregarding the constrained cleaning availability constraint, but this did lead to numerous constraint violations for all the final solutions.

Chapter 7

Conclusion

In this report, we have explored the mathematical field of scheduling theory, with our main focus being on practical applications in production settings. We have seen how formal definitions and classifications can bring structure to complex scheduling environments, giving insight in how to approach these problems. Additionally, we discussed how we can deal with practical constraints that arise in real-life production settings. To this end, we also introduced a simple framework on how we can classify and deal with such constraints, that are not always encountered in literature. Finally, we introduced some solution methods that find common use in scheduling problems, along with both advantages and disadvantages of some of these methods in different situations.

To further clarify all these methods and to prove their effectiveness, we have dealt with a case study of a real manufacturer of generic medicine and explained how we applied the different parts of theory to our problem. In the end, we created a mathematical tool capable of generating strong feasible schedules for the company, which is now being used. The schedules generated by this method, even when there are significant areas of potential improvement, are strong enough that a human or simple computer-driven tools cannot realistically provide solutions of similar quality, let alone in the same amount of time. When comparing the solutions obtained by our created tool provided a decrease of total tardiness at least 30% when compared to simple dispatching rules, and a decrease of at least 10% when compared to the previously used methods by the company. We have thus both provided a guide on how to overcome certain difficulties that may arise when trying to apply this theory in practice, as well as having showcased a concrete example in practice, serving as a proof of concept.

While these results are great in and of itself, even better results may very well be possible in relatively simple ways. One example of this is with the tuning of our final model, where we tested a handful of alternatives, and in some cases found surprising results. In our case, we only did limited testing and not all possible alternatives were included. While it is likely that repeated finetuning will give diminishing results, one can only be sure by actually trying it out. In a similar manner, other solution methods might have proven to be more effective.

Due to the complexity of problems, it is not always clear what methods perform better, but seeing as development time is often limited, we have gladly settled for 'good', instead of endlessly searching for 'perfect'.

Bibliography

[1] H. Van Dyke Parunak. "Characterizing the manufacturing scheduling problem". In: *Journal of Manufacturing Systems* 10.3 (1991), pp. 241–259. ISSN: 0278-6125. DOI: https://doi.org/10.1016/0278-6125(91)90037-3. URL: https://www.sciencedirect.com/science/article/pii/0278612591900373.

[2] Rodrigo Romero-Silva, Javier Santos, and Margarita Hurtado. "A conceptual framework of the applicability of production scheduling from a contingency theory approach: addressing the theory-practice gap". In: *Production Planning and Control* (May 2022), pp. 1–21. DOI: 10.1080/09537287.2022.2076627.

[3] Sundus Shukar et al. "Drug Shortage: Causes, Impact, and Mitigation Strategies". In: *Frontiers in Pharmacology* 12 (2021). ISSN: 1663-9812. DOI: 10.3389/fphar.2021.693426. URL: https://www.frontiersin.org/journals/pharmacology/articles/10.3389/fphar.2021.693426.

[4] Eduardo Guzman, Beatriz Andres, and Raul Poler. "Models and algorithms for production planning, scheduling and sequencing problems: A holistic framework and a systematic review". In: *Journal of Industrial Information Integration* 27 (2022), p. 100287. ISSN: 2452-414X. DOI: https://doi.org/10.1016/j.jii.2021.100287. URL: https://www.sciencedirect.com/science/article/pii/S2452414X21000844.

[5] J. Mula et al. "Models for production planning under uncertainty: A review". In: *International Journal of Production Economics* 103.1 (2006), pp. 271–285. ISSN: 0925-5273. DOI: https://doi.org/10.1016/j.ijpe.2005.09.001. URL: https://www.sciencedirect.com/science/article/pii/S0925527306000041.

[6] Song Ye et al. "Efficient heuristic for solving non-permutation flow-shop scheduling problems with maximal and minimal time lags". In: *Computers Industrial Engineering* 113 (2017), pp. 160–184. ISSN: 0360-8352. DOI: https://doi.org/10.1016/j.cie.2017.08.024. URL: https://www.sciencedirect.com/science/article/pii/S0360835217303832.

[7] Ali Hasani, Seyed Mohammad Hassan Hosseini, and Shib Sankar Sana. "Scheduling in a flexible flow shop with unrelated parallel machines and machine-dependent process stages: Trade-off between Makespan and production costs". In: *Sustainability Analytics and Modeling* 2 (2022), p. 100010.

[8] Jitti Jungwattanakit et al. "An Evaluation of Sequencing Heuristics for Flexible Flowship Scheduling Problems with Unrelated Parallel Machines and Dual Criteria". In: (Jan. 2007).

[9] Mohammad Saidi-Mehrabad and Parviz Fattahi. "Flexible job shop scheduling with tabu search algorithms". In: *The International Journal of Advanced Manufacturing Technology* 32 (2007), pp. 563–570.

[10] R.L. Graham et al. "Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey". In: *Discrete Optimization II*. Ed. by P.L. Hammer, E.L. Johnson, and B.H. Korte. Vol. 5. Annals of Discrete Mathematics. Elsevier, 1979, pp. 287–326. DOI: https://doi.org/10.1016/S0167-5060(08)70356-X. URL: https://www.sciencedirect.com/science/article/pii/S016750600870356X.

[11] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. "Complexity of Machine Scheduling Problems". In: *Studies in Integer Programming*. Ed. by P.L. Hammer et al. Vol. 1. Annals of Discrete Mathematics. Elsevier, 1977, pp. 343–362. DOI: https://doi.org/10.1016/S0167-5060(08)70743-X. URL: https://www.sciencedirect.com/science/article/pii/S016750600870743X.

[12] Fadi Shrouf et al. "Optimizing the production scheduling of a single machine to minimize total energy consumption costs". In: *Journal of Cleaner Production* 67 (2014), pp. 197–207. ISSN: 0959-6526. DOI: https://doi.org/10.1016/j.jclepro.2013.12.024. URL: https://www.sciencedirect.com/science/article/pii/S0959652613008780.

[13] Iman Rahimi et al. "A Review on Constraint Handling Techniques for Population-based Algorithms: from single-objective to multi-objective optimization". In: *Archives of Computational Methods in Engineering* 30 (2023), pp. 2181–2209.

[14] Steven Vajda. *Mathematical programming*. Courier Corporation, 2009.

[15] E. L. Lawler and D. E. Wood. "Branch-and-Bound Methods: A Survey". In: *Operations Research* 14.4 (1966), pp. 699–719. DOI: 10.1287/opre.14.4.699. eprint: https://doi.org/10.1287/opre.14.4.699. URL: https://doi.org/10.1287/opre.14.4.699.

[16] Pavol Semanco and Vladimir Modrak. "A comparison of constructive heuristics with the objective of minimizing makespan in the flow-shop scheduling problem". In: *Acta Polytechnica Hungarica* 9.5 (2012), pp. 177–190.

[17] Fatos Xhafa and Ajith Abraham. *Metaheuristics for scheduling in industrial and manufacturing applications*. Vol. 128. Springer Science & Business Media, 2008.

Appendix A

First Stage Model

In this appendix, we take a look at the exact model formulation of our first model, and explain every aspect of it along the way. The three-field notation of this model was as follows: $F \mid r_j, S_{ij} \mid \sum w_j T_j$. In order to turn this into a mathematical model, we have to 'translate' each rule, associated with what a schedule may or may not look like, into some mathematical expression. These mathematical expressions of course need some variables that they depend on, which we call *decision variables* in our context. These decision variables consist of the aspects of a schedule we wish to decide, in other words, the values of which we are interested in their optimal values. For our scheduling problem, this is simply the starting times of each job at each stage, combined with the machine that job is assigned to at that stage. After all, if we know all the starting times of our jobs and the machines that process them, the rest of the schedule simply follows from there.

Now, we also wish to add constraints that form relations between our decision variables. One example for such a relation would be that the starting time of some job at the third production stage, cannot be before the starting time of that same job at the second stage plus the processing time for that job at the second production stage. Or in simpler words, a job can only start an operation when its previous operation is completed. In mathematical terms, this constraint would look like this:

$$t_{js} \geq t_{js_{previous}} + P_{js_{previous}}$$

As you can tell, we are using some notation here we have not seen previously. You may be able to decipher the meaning of this notation from the context, but we should really just properly define it, which is another important part of our model formulation. Above the following notations have been used:

- t_{js} denotes the starting time of job j at stage s.
- P_{js} denotes the processing time of job j at stage s.
- $s_{previous}$ refers to the stage immediately before s.

When defining our model like this, we may run into some problems that need revision. For example, we just defined P as the processing time of a job at a certain stage, but this is not really accurate, since the processing time at a stage can be different depending on the machine we are using. Therefore, in our case we would have to modify this as follows:

- P_{jk} denotes the processing time of job j at machine k.
- a_{jk} is a binary variable that equals 1 if job j is assigned to machine k and equals 0 otherwise.

We can now modify our constraint as well.

$$t_{js} \geq t_{js_{previous}} + \sum_{k \in M_{js_{previous}}} a_{jk} \cdot P_{jk}$$

We introduced one more piece of notation, namely M, this is the set of all machines, and M_{js} then denotes the set of machines at stage s that are able to process job j. The last part of our constraint then denotes the summation of $a_{jk} \cdot P_{jk}$ for all values of k in M_{js}, or in simpler terms, it sums the processing time of our job j for each possible machine at stage s where j can be processed, but it only counts them when job j is actually assigned to a particular machine, since a_{jk} is 0 otherwise, and the product would be 0 as well.

We have currently written our constraint specifically for job j, but this constraint must hold for every single job in our scheduling problem. To account for this, without having to write hundreds of constraints, we use the following notation at the end of our constraint.

$$t_{js} \geq t_{js_{previous}} + \sum_{k \in M_{js_{previous}}} a_{jk} \cdot P_{jk} \quad \forall j \in J, s \in S/\{s_{first}\}$$

What this is currently saying, is that our constraint must hold for every job j that is part of the set of all jobs J and for every stage s that is part of the set of all stages S, with the exception of s_{first}. The reason for this exception is that the constraint makes no sense for a job in the first stage, as a previous stage does not exist. Using this method of defining multiple constraints at once, we can keep our formulation relatively concise.

We could follow a similar process as above for the other constraints, such as the release dates for our jobs and how we should incorporate the objective function. But for the sake of brevity we skip those and give the full formulation, followed by some explanations regarding each set of constraints.

Parameters:

- S: The set of stages. ($S = \{weigh, granulate, tabletting, coating, packaging\}$)
- J: The set of jobs (=product orders).

- M: The set of machines.
- M_{js}: The set of machines in stage $s \in S$.
- P_{jk}: Processing time of job j on machine k.
- r_j: The release date of job j (=the earliest time we can schedule a job).
- d_j: The due date of job j (=the date the order is supposed to be ready for pickup).
- ST_{ijk}: The setup time between jobs i and j on machine k.
- Q: Some large enough number.
- w_j: The weight of job j.

Variables:

- t_{js}: The starting time of job j at stage s.
- C_j: The completion time of job j.
- T_j: The tardiness of job j.
- L_j: The lateness of job j.
- b_{ijs}: Binary variable equal to 1 when job i is scheduled before job j at stage s and 0 otherwise.
- a_{jk}: Binary variable equal to 1 when job j is assigned to machine k and 0 otherwise.

Model:

$$\min \sum_{j \in J} w_j T_j$$

s.t.

$$T_j \geq L_j \qquad \forall j \in J \tag{A.1}$$

$$L_j = C_j - d_j \qquad \forall j \in J \tag{A.2}$$

$$C_j = t_{js_{last}} + \sum_{k \in M_{js_{last}}} a_{jk} \cdot P_{jk} \qquad \forall j \in J \tag{A.3}$$

$$\sum_{k \in M_{js}} a_{jk} = 1 \qquad \forall j \in J, s \in S \tag{A.4}$$

$$t_{js} \geq t_{js_{previous}} + \sum_{k \in M_{js_{previous}}} a_{jk} \cdot P_{jk} \qquad \forall j \in J, s \in S/\{s_{first}\} \tag{A.5}$$

$$t_{js_{first}} \geq r_j \qquad \forall j \in J \tag{A.6}$$

$$t_{js} \geq t_{is} + P_{ik} + ST_{ijk} - Q \cdot (2 - a_{jk} - a_{ik}) - Q \cdot (1 - b_{ijs}) \qquad \forall i, j \in J, s \in S, k \in M_s \tag{A.7}$$

$$b_{ijs} + b_{jis} = 1 \qquad \forall i, j \in J, s \in S \tag{A.8}$$

$$t_{js}, C_j, T_j, L_j \geq 0 \qquad \forall j \in J, s \in S \tag{A.9}$$

$$b_{ijs}, ajk \in \{0, 1\} \qquad \forall i, j \in J, s \in S, k \in M \tag{A.10}$$

In constraints (A.1) - (A.3), we define the objective function to be the total tardiness over all jobs, and to this end define C, L, and T. Constraints (A.4) ensure every job is assigned to precisely one machine in each production stage. Constraints (A.5) make sure that a job can only start a new operation when its previous operation has finished. Constraints (A.6) represent every job only being able to start the first operation at or after its release date. In constraints (A.7) and (A.8), we ensure that when two different jobs are to be processed on the same machine, one of them is forced to be processed first, making sure they are not processed simultaneously. Lastly, constraints (A.9) ensure the

non-negativity of t, C, T and L, and constraints (A.10) ensure that a and b are indeed binary.

Two things to note are the exact workings of the first three sets of constraints and the addition of the Q parameter in constraints (A.7). Constraints (A.2) and (A.3) simply define L_j and C_j. Combining this with constraints (A.1) and the non-negativity of T_j, we obtain that T_j is larger than or equal to both L_j and 0, and thus $T_j \geq \max(L_j, 0)$. Finally, since we are minimizing the sum of all T_j's, they must be exactly equal to $\max(L_j, 0)$ in any optimal solution.

The addition of the Q parameter in constraints (A.7) is the result of a commonly used trick in model descriptions to keep all the constraints linear in the variables. This is common practice because having linear constraints leads to some very desirable qualities of the model, that we return to when talking about solution methods.

Now that a basic model has been established, it can be tested against real life data. Unfortunately, in our case this quickly ran into a number of problems, which will be outlined in the second half of Chapter 3.

Appendix B

Simulated Annealing

In this appendix, we give a more detailed explanation of the Simulated Annealing (SA) metaheuristic. This consists of the following steps:

1. In SA, we start with an initial solution σ and an initial temperature T.
2. We continue by generating a new solution σ', by making a small modification to the current solution. This is also referred to as generating a *neighbor* solution.
3. Next we compare the objective value of both our current solution v and the neighbor solution v'. If v' is better than v, we update our current solution to become σ'. If v' is worse than v, we determine some probability P that we update our current solution to become σ', depending on v, v', and T.
4. After determining whether or not we update our solution or not, we update our temperature following some predefined cooling schedule.
5. Next we check some sort of termination condition. If the condition is met, we terminate and have obtained our final solution. If the condition is not yet met, we return to step 2.

A common termination condition is simply a maximum runtime or maximum number of iterations. For the cooling schedule, we can then use the following:

$$T = T_0 \cdot \frac{t_{max} - t}{t_{max}}$$

Where T_0 is the initial temperature, t is the current elapsed time and t_{max} is the maximum runtime we use. Lastly, we can use the following formula for P:

$$P = e^{-\frac{\Delta v}{T}}$$

Where Δv is the difference between v and v'.

Appendix C

Tabu Search

In this appendix, we give a more detailed explanation of the Tabu Search (TS) metaheuristic. This consists of the following steps:

1. In TS, we start with an initial solution σ and an empty *tabu list*.
2. We continue by generating a number of new solutions $\sigma'_1, \ldots, \sigma'_n$, by making a small modification to the current solution.
3. Out of the generated solutions, we compare the objective value of each of them, and pick the best solution, σ'_*, that is not currently in our tabu list, from among them. We then update our current solution to become σ'_*.
4. We continue by adding our previous solution to the tabu list.
5. Next we check some sort of termination condition, if the condition is met, we terminate and have obtained our final solution. If the condition is not yet met, we return to step 2.

Similar to simulated annealing, a common termination condition is simply a maximum runtime or number of iterations. An additional thing to keep in mind, is the size of our tabu list. If we were to run this algorithm for hundreds of thousands of iterations, the tabu list becomes very large, and constantly checking whether potential solutions are inside this list becomes computationally expensive. To deal with this problem, it is common to have a maximum size for the tabu list. Once we exceed this maximum in step 4, we simply remove the element that first entered the list before continuing with step 5.

Appendix D

Solution Quality over Time

In this appendix, we provide graphs showcasing the progression of solution quality for the simulated annealing algorithm at different initial temperatures. We can see a clear convergence for all of these.

Figure D.1: Objective value over time when using simulated annealing with an initial temperature of 1.

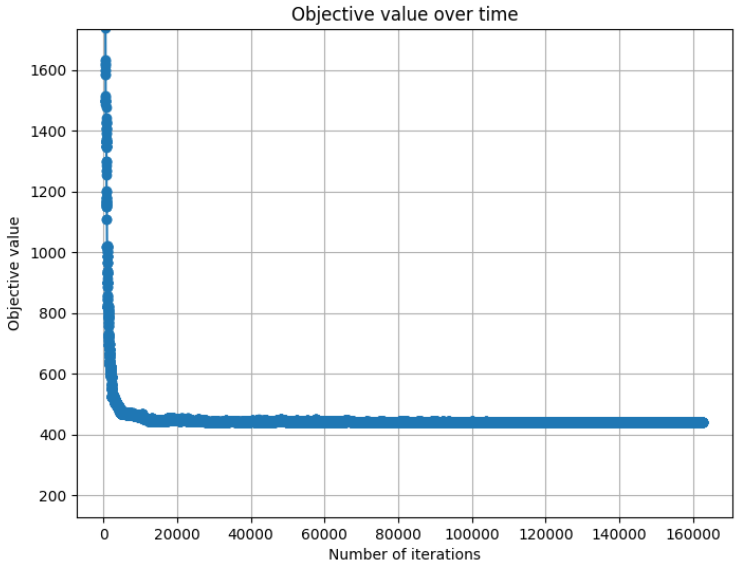


Figure D.2: Objective value over time when using simulated annealing with an initial temperature of 5.

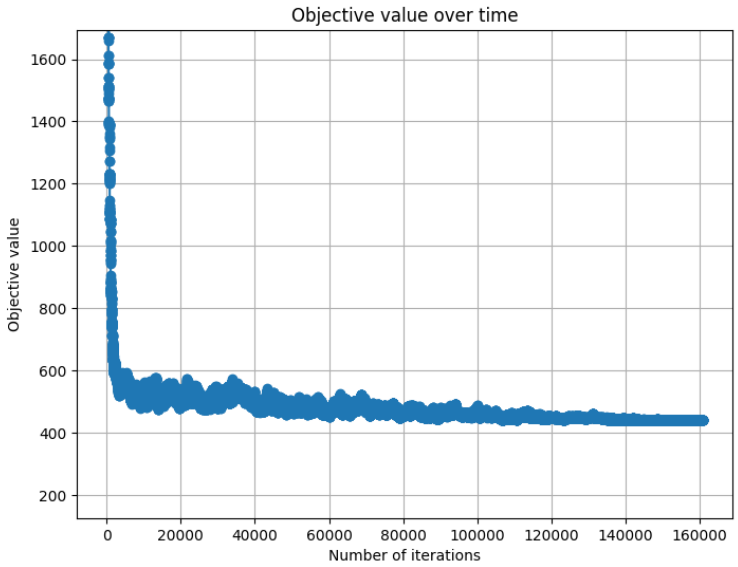


Figure D.3: Objective value over time when using simulated annealing with an initial temperature of 10.

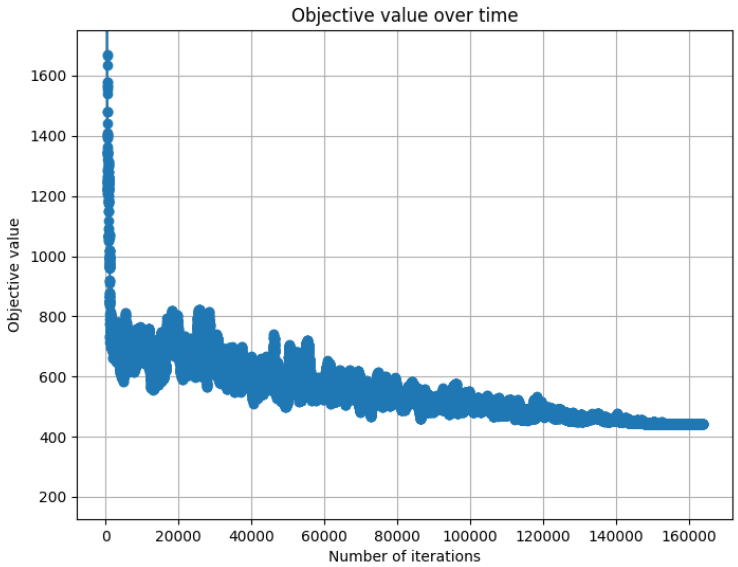
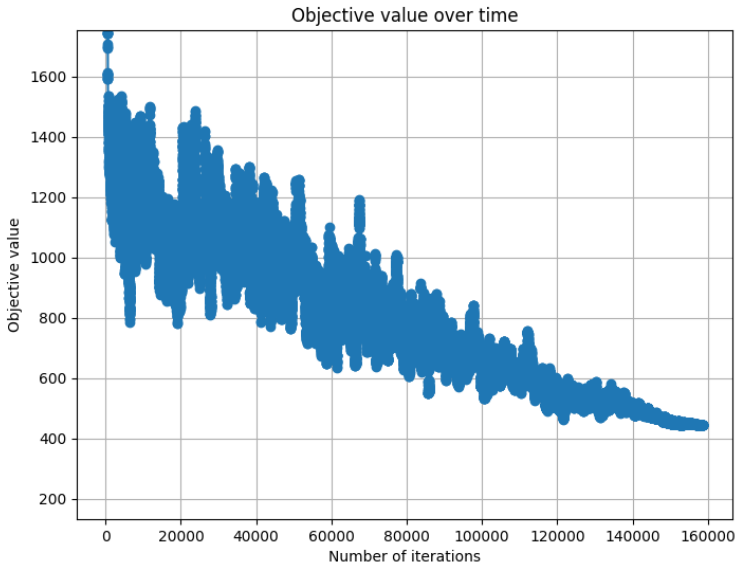


Figure D.4: Objective value over time when using simulated annealing with an initial temperature of 25.



Appendix E

Solution Evaluation Algorithm

The aim of this section is to give an overview of the solution evaluation algorithm used in the final solution approach of the case study. To this end, we first outline the form of our input and output:

- **Input.** Our input is given in the form of a sequence of jobs for each stage, specifying in what order jobs are processed in a particular stage. This is combined with a list of assigned machines for each jobs, specifying for each stage, what machine the job will be processed on.
- **Output.** As our output, simply using the objective value does suffice, as it is the only part we need to use in our simulated annealing approach. When a final solution is obtained we can simply rerun this algorithm to obtain the details of the solution.

Our algorithm consists of the following steps:

1. **Job Selection.** We construct our solution stage by stage in the order given by our job sequence. This means we start by considering what stage we are currently in, followed by picking the first job in that stage's sequence we have no yet scheduled in this stage. Once a stage has been completely scheduled, we can move on to the following stage.
2. **Job Placement.** When we know what job we are sequencing, we simply consider the specific machine it has been assigned to and determine the earliest possible time this job can be started at that machine while not breaching any constraints. This means we only consider jobs that were already placed in the schedule, which are the ones earlier in our sequence.
3. **Objective Evaluation.** When all jobs have been placed, we can simply evaluate the schedule by computing the total weighted tardiness and testing for any of the penalty-based constraints.

The method we use during the job placement step in order to determine the earliest possible time a job j can be started at a particular machine, is by simply considering two things: What is the earliest time the machine is available to process, and what is the earliest time the job is available to be processed. We refer to these times as the *machine availability* and *job availability*, and will shortly go over both of them:

- The machine availability, where we first determine what type of cleaning must happen between this job and the previous one processed by the same machine. This is also where the Periodic Cleaning constraint comes in. After we know what type of cleaning is necessary, we can compute the completion time of the previous job processed by this machine, add the cleaning time, resulting in the earliest time the machine is available.
- The job availability, where we determine the time this job can start at this stage, which is either the completion time of the previous stage (potentially plus some buffer time), or in case of no previous stage the release date r_j of this job.

The job's starting time is then the maximum of these two quantities.